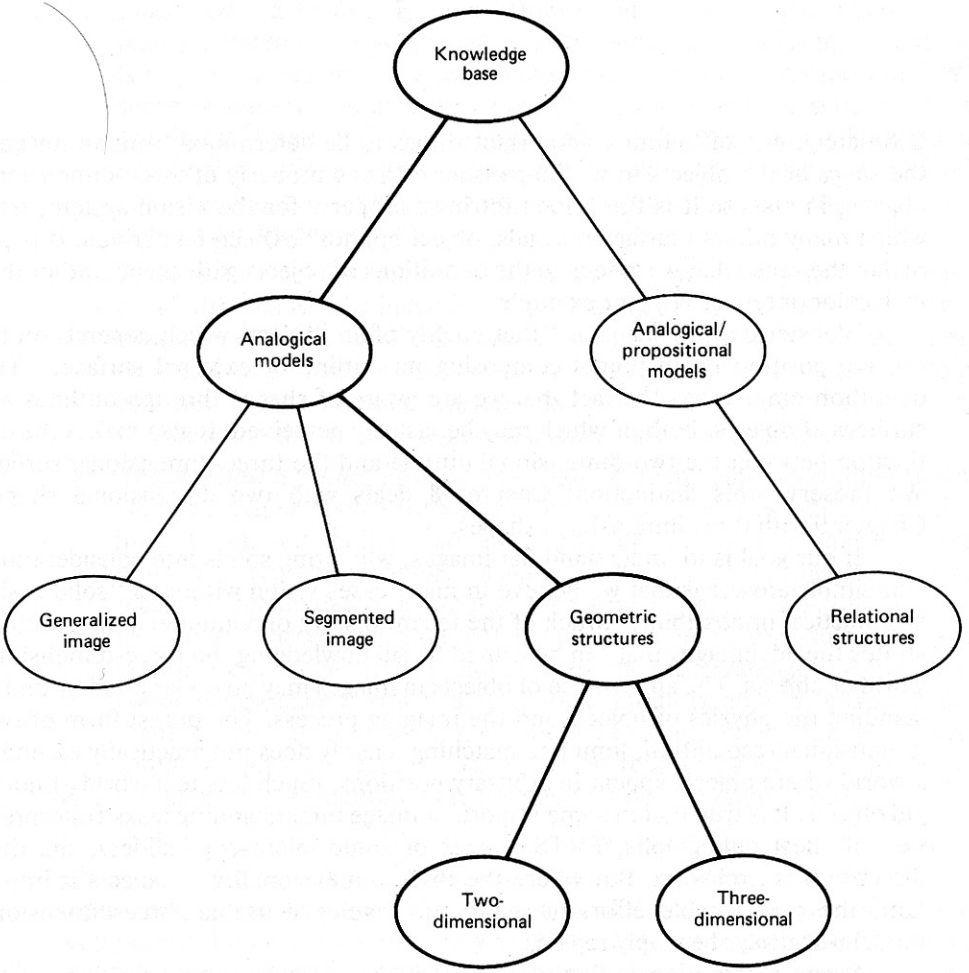
GEOMETRICAL STRUCTURES III



Ultimately, one of the most important things to be determined from an image is the *shape* of the objects in it. Shape is an intrinsic property of three-dimensional objects; in a sense it is the primal intrinsic property for the vision system, from which many others (surface normals, object boundaries) can be derived. It is primal in the sense that we associate the definitions of objects with shape, rather than with color or reflectivity, for example.

Webster defines shape as "that quality of an [object] which depends on the relative position of all points composing its outline or external surface." This definition emphasizes the fact that we are aware of shapes through outlines and surfaces of objects, both of which may be visually perceived. It also makes the distinction between the two-dimensional outline and the three-dimensional surface. We preserve this distinction: Chapter 8 deals with two dimensional shapes, Chapter 9 with three dimensional shapes.

If our goal is to understand flat images, why bring solids into consideration? Our simple answer is that we believe in many cases vision without a "solid basis" is a practical impossibility. Much of the recent history of computer vision demonstrates the advantages that can be gained by acknowledging the three-dimensional world of objects. The appearance of objects in images may be understood by understanding the physics of objects and the imaging process. The purest form of two-dimensional recognition, template matching, clearly does not practically extend to a world where objects appear in arbitrary positions, much less to a world of nonrigid objects. It is true that in some important image understanding tasks (interpretation of chest radiographs, ERTS images or some microscope slides), the third dimension is irrelevant. But where the three-dimensionality of objects is important, the considerable effort necessary to develop a usable three-dimensional model will always be amply repaid.

Shape recognition is doubtless one of the most important facilities of the mammalian visual system. We have seen how important shape information can be

extracted from images in early processing and segmentation. One of the major challenges to computer vision is to represent shapes, or the important aspects of shapes, so that they may be learned, matched against, recollected, and used. This effort is hampered by several factors.

1. *Shapes are often complex.* Whereas color, motion, and intensity are relatively simply quantified by a few well-understood parameters, shape is much more subtle. Common manufactured or natural shapes are incredibly complex; they may be represented "explicitly" (say by representing their surface) only with hundreds of parameters. Worse, it is not clear what aspects of shapes are important for applications such as recognition. An explicit and complete representation may be computationally intractable for such basic uses as matching. What "shape features" can be used to ease the burden of computation with complex shapes?
2. *Introspection is no help.* Human beings seem to have a large fraction of their brains devoted to the single task of shape recognition. This important activity is largely "wired in" at a level below our conscious introspection. Why is shape recognition so easy for human beings and shape description so hard? The fact that we have no precise language for shape may argue for the inaccessibility of our shape-processing algorithms or data structures. This lack of cognitive leverage is a trifle daunting, especially when taken with the complexity of everyday shapes.
3. *There is little classical guidance.* Mathematics traditionally has not concerned itself with shape. For instance, only recently has there been a mathematical definition of "rigid solid" that accords with our intuition and of set operations on solids that preserve their solidity. The fact that such basic questions are only now being addressed indicates that computer science must do more than encode some already existing proven ideas. Thus we have the next point.
4. *The discipline is young.* Until very recently, human beings communicated about complex shapes mainly through words, gestures, and two-dimensional drawings. It was not until the advent of the digital computer that it became of interest to represent complex shapes so that they could be specified to the machine, manipulated, computed with, and represented as output graphics. No generally accepted single representation scheme is available for all shapes; several exist, each with its advantages and disadvantages. Algorithms for manipulating shapes (for example, for computing how to move a sofa up a flight of stairs, or computing the volume of a specified shape) are surprisingly complex, and are research topics. Often the representations good for one application, such as recognition, are not good for other computations.

It is the intention of this part of the book to indicate some of what is known about the representation of shape. Although the details of geometric representations may be still under development, they are an essential part of our layered computer vision organization. They are more abstract than segmented structures and are distinguished from relational structures by their preponderance of metric information.

Representation of Two-Dimensional Geometric Structures 8

8.1 TWO-DIMENSIONAL GEOMETRIC STRUCTURES

The structures of this chapter are the intuitive ones of well-behaved planar regions and curves. A mathematical characterization of these structures that bars "pathological" cases (such as regions of a single point and space-filling curves) is possible [Requicha 1977]. Basically the requirement is that regions be "homogeneously two-dimensional" (contain no hanging or isolated structures of different dimension—solids, lines or points). Similarly, curves should be homogeneously one-dimensional. The property of regularity is sometimes important; a regular set is one that is the closure of its interior (in the relevant one- or two-dimensional topology). Intuitively, regularizing a two-dimensional set (taking the closure of its interior) first removes any hanging one- and zero-dimensional parts, then covers the remainder with a tight skin (Fig. 8.1). In computer vision, often regions and curves are discrete, being defined on a raster of pixels or on an orthogonal grid of possible primitive edge segments. It is frequently convenient to associate a direction with a curve, hence ordering the points along it and defining portions of the plane to its left and right.

The one-dimensional closed curve that bounds a well-behaved region is an unambiguous representation of it; Section 8.2 deals with representations of curves and hence indirectly of regions. Section 8.3 deals with other unambiguous representations of regions that are not based on the boundary. Sometimes unambiguous representation is not the issue; it may be important to have qualitative description of a region (its size or shape, say). Section 8.4 presents several terse descriptive properties for regions.

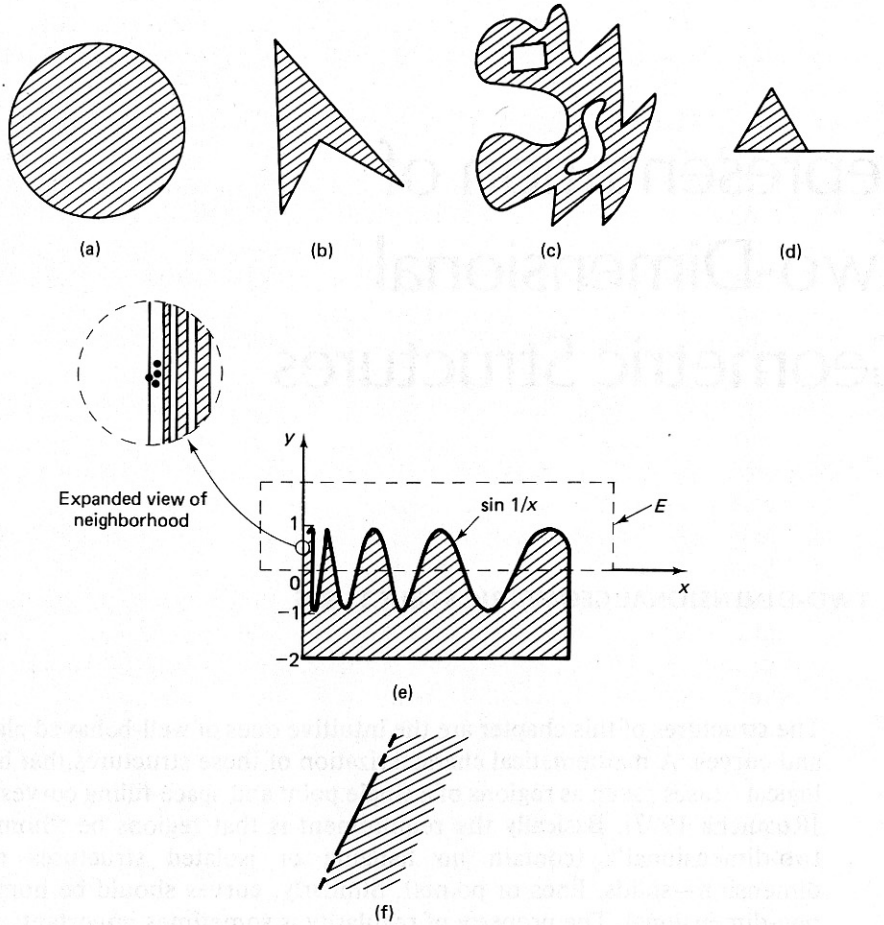


Fig. 8.1 (a, b, c) are Regions; (d) (e) and (f) are not.

8.2 BOUNDARY REPRESENTATIONS

8.2.1 Polylines

The "two-point" form of a line segment (see Appendix 1) extends easily to the *polyline*, which represents a concatenation of line segments as a list of points. Thus the point list $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ represents the concatenation of the line segments from $\mathbf{x}_1$ to $\mathbf{x}_2$ and from $\mathbf{x}_2$ to $\mathbf{x}_3$. If the first point is the same as the last, a closed boundary is represented.

Polylines can approximate most useful curves to any desired degree of accuracy. One might think there is one obvious way to approximate a boundary curve (or raw data) with a polygonal line. This is not so: many different approaches are possible. Finding a satisfying polygonal approximation to a given curve basically involves segmentation issues. The problem is to find corners or *breakpoints* that

yield the "best" polyline. As with region-based segmentation schemes, the ideas here can be characterized by the concepts of *merging* and *splitting*. Splitting and merging schemes may be combined, especially if the appropriate number of linear segments is known beforehand. For details, see [Horowitz and Pavlidis 1976].

In a merging algorithm, points along a curve (possibly in image data) are considered in order and accepted into a linear segment as long as they fit sufficiently well. When they do not, a new segment is begun. The efficiency and characteristics of these schemes are quite variable, and endless variations on the general idea are possible. A few examples of "one pass" merging schemes are given here: explicit algorithms are available in [Pavlidis 1977].

If the boundary (represented on a discrete grid) is known to be piecewise linear, it is specified by its breakpoints. To find them, one can look along the boundary, monitoring the angle between two line segments. One segment is between the current point and a point several points back along the boundary; the other is between the current point and one several points forward. When the angle between these segments reaches a maximum over some threshold, a breakpoint is declared at the current point. This scheme does not adjust breakpoint positions, and so is fast [Shirai 1975] but works best for piecewise linear input curves.

Tolerance-band solutions place a point on either side of the curve at the maximum allowable error distance, and then find the longest piece of the curve that lies entirely between parallel lines through the two points [Tomek 1974]. This method proceeds without breakpoint adjustment, and may not find the most economical set of segments (Fig. 8.2).

An approximation of a curve with a polyline of minimum length in error by at most a pixel is given in [Sklansky and Kibler 1976]. Each curve pixel is considered a square and the resulting pixel structure is four-connected. The approximation describes the shape of an elastic thread placed in the pixel structure (Fig. 8.3). The

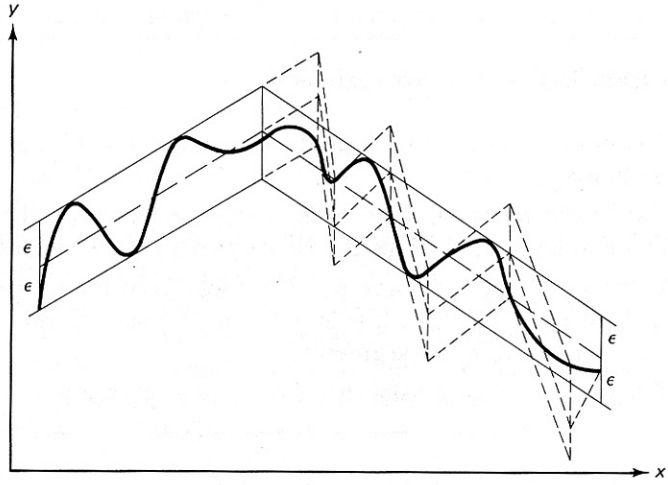


Fig. 8.2 Simple tolerance-band solution (dotted lines). Better solution (solid lines).

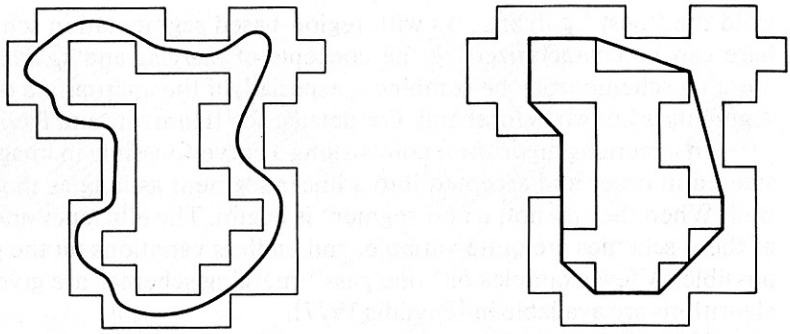


Fig. 8.3 Minimum length polyline.

method tends to have difficulties with curves that are sharp relative to the grid size.

Another scheme, [Roberts 1965] is to keep a running least-squared-error best-fit line calculation for points as they are merged into segments [Appendix 1]. When the residual (error) of a point goes over some threshold or the accumulated error for a segment exceeds a threshold, a new segment is started. Difficulties arise here because the concept of a breakpoint is nonexistent; they just occur at the intersections of the best-fit lines, and without a phase of adjusting the set of points to be fit by each line (analogous to breakpoint adjustment), they may not be intuitively appealing.

Generally, one-pass merging schemes do not produce the most satisfying polylines possible under all conditions. Part of the problem is that breakpoints are only introduced after the fit has deteriorated, usually indicating that an earlier breakpoint would have been desirable.

In a *splitting* scheme, segments are divided (usually into two parts) as long as they fail some fitting condition [Duda and Hart 1973; Turner 1974]. Algorithm 8.1 provides an example.

Algorithm 8.1: Curve Approximation

1. Given a curve as in Fig. 8.4a, draw a straight line between its end points (Fig. 8.4b).
2. For every point on the curve, compute its perpendicular distance to the approximating (poly)line. If it is everywhere within some tolerance, exit.
3. Otherwise, pick the curve point farthest from the approximating (poly)line, make it a new breakpoint (Fig. 8.4c) and replace the relevant segment of polyline with two new line segments.
4. Recursively apply the algorithm to the two new segments (Fig. 8.4d).

A straightforward extension is needed to deal with the case of curve segments parallel to the approximating one at maximum distance (Fig. 8.4e).

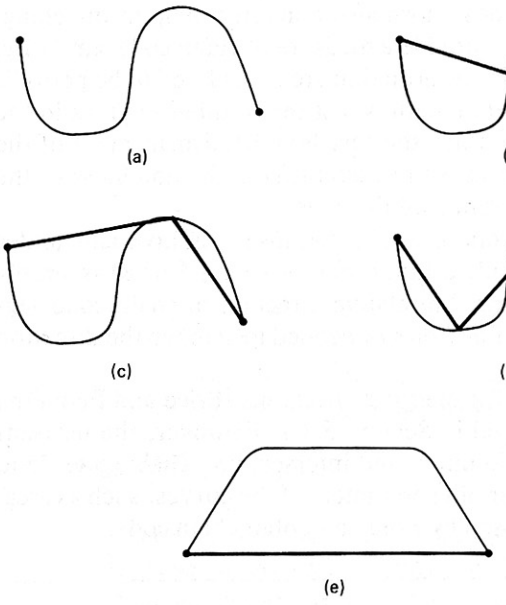


Fig. 8.4 Stages in the recursive linear segmenter (see text).

The area of a polygon may easily be computed from its polyline representation [Roberts 1965]. For a closed polyline of n points $(x(i), y(i))$, $i=0, \ldots, n-1$, labeled clockwise around a polygonal boundary, the area of the polygon is

$$\frac{1}{2} \sum_{i=0}^{n-1} (x_{i+1} y_i - x_i y_{i+1}) \tag{8.1}$$

where subscript calculations are modulo n. This formula can be proved by considering it as the sum of (signed) areas of triangles, each with a vertex at the origin, or of parallelograms constructed by dropping perpendiculars from the polyline points to an axis. This method specializes to chain codes, which are a limiting case of polylines.

8.2.2 Chain Codes

Chain codes [Freeman 1974] consist of line segments that must lie on a fixed grid with a fixed set of possible orientations. This structure may be efficiently represented because of the constraints on its construction. Only a starting point is represented by its location; the other points on a curve are represented by successive displacements from grid point to grid point along the curve. Since the grid is uniform, direction is sufficient to characterize displacement. The grid is usually considered to be four- or eight- connected; directions are assigned as in Fig. 8.5, and each direction can be represented in 2 or 3 bits (it takes 18 bits to represent the starting point in a 512×512 image).

Chain codes may be made position-independent by ignoring the "start point." If they represent closed boundaries they may be "start point normalized" by choosing the start point so that the resulting sequence of direction codes forms

an integer of minimum magnitude. These normalizations may help in matching. Periodic correlation (Section 3.2.1) can provide a measure of chain code similarity. The chain codes without their start point information are considered to be periodic functions of "arc length." (Here the arc length is just the number of steps in the chain code.) The correlation operation finds the (arc length) displacement of the functions at which they match up best as well as quantifying the goodness of the match. It can be sensitive to slight differences in the code.

The "derivative" of the chain code is useful because it is invariant under boundary rotation. The derivative (really a first difference mod 4 or 8) is simply another sequence of numbers indicating the relative direction of chain code segments; the number of left hand turns of $\pi/2$ or $\pi/4$ needed to achieve the direction of the next chain segment.

Chain codes are also well-suited for merging of regions [Brice and Fennema 1970] using the data structure described in Section 5.4.1. However, the pleasant properties for merging do not extend to union and intersection. Chain codes lend themselves to efficient calculation of certain parameters of the curves, such as area. Algorithm 8.2 computes the area enclosed by a four-neighbor chain code.

Algorithm 8.2: Chain Code Area

Comment: For a four-neighbor code (0: $+x$, 1: $+y$, 2: $-x$, 3: $-y$) surrounding a region in a counterclockwise sense, with starting point (x, y):

```
begin Chain Area;
1. area := 0;
2. yposition := y;
3. For each element of chain code
   case element-direction of
   begin case
     [0] area := area-yposition;
     [1] yposition := yposition + 1;
     [2] area := area + yposition;
     [3] yposition := yposition - 1;
   end case;
end Chain Area;
```

To merge two region boundaries is to remove any boundary they share, obtaining a boundary for the region resulting from gluing the two abutting regions together. As we saw in Chapter 5, the chain codes for neighboring regions are closely related at their common boundary, being equal and opposite in a clearly defined sense (for N-neighbor chain codes, one number is equal to the other plus $N/2$ modulo N (see Chapter 5). This property allows such sections to be identified readily, and easily scissored out to give a new merged boundary. As with polylines, it is not immediately obvious from a chain-coded boundary and a point whether the point is within the boundary or outside. Many algorithms for use with chain code representations may be found in [Freeman 1974; Gallus and Neurath 1970].

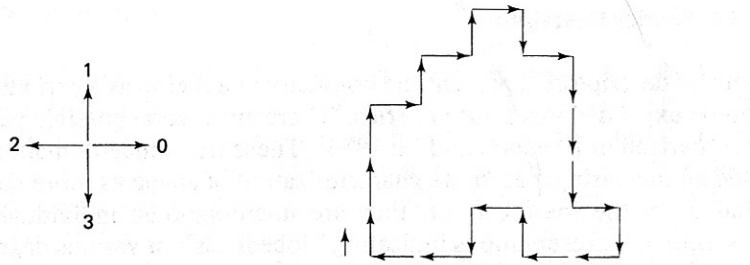


Fig. 8.5 (a) Direction numbers for chain code elements. (b) Chain code for the boundary shown. (c) Derivative of (b).

8.2.3 The ψ -s Curve

The $\psi - s$ curve is like a continuous version of the chain code representation; it is the basis for several measures of shape. ψ is the angle made between a fixed line and a tangent to the boundary of a shape. It is plotted against s, the arc length of the boundary traversed. For a closed boundary, the function is periodic, with a discontinuous jump from 2π back to 0 as the tangent reattains the angle of the fixed line after traversing the boundary.

Horizontal straight lines in the $\psi - s$ curve correspond to straight lines on the boundary (ψ is not changing). Nonhorizontal straight lines correspond to segments of circles, since ψ is changing at a constant rate. Thus the $\psi - s$ curve itself may be segmented into straight lines [Ambler et al. 1975], yielding a segmentation of the boundary of the shape in terms of straight lines and circular arcs (Fig. 8.6).

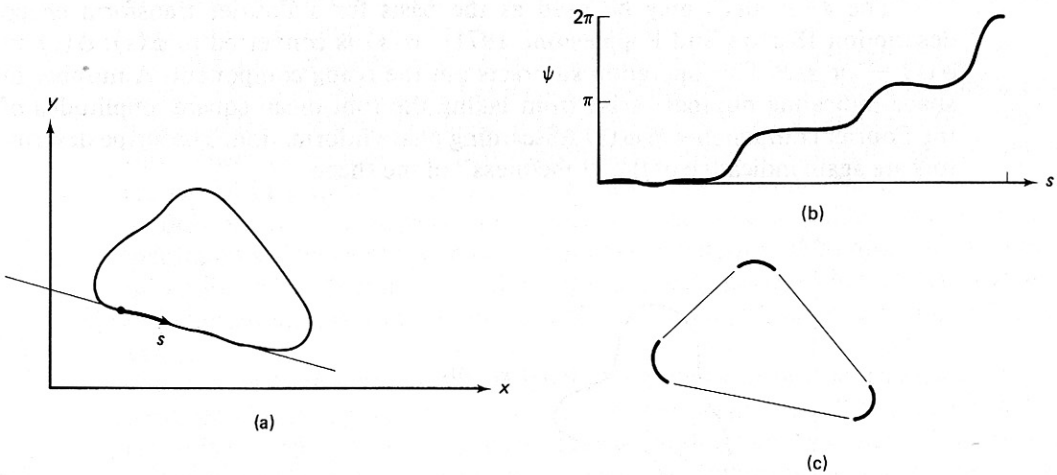


Fig. 8.6 ψ-s segmentation. (a) Triangular curve and a tangent. (b) ψ-s curve showing regions of high curvature. (c) Resultant segmentation.

8.2.4 Fourier Descriptors

Fourier descriptors represent the boundary of a region as a periodic function which can be expanded in a Fourier series. There are several possible parameterizations, summarized in [Persoon and Fu 1974]. These frequency-domain descriptions provide an increasingly accurate characterization of shape as more coefficients are included. In the infinite limit, they are unambiguous; individual coefficients are descriptive representations indicating "lobedness" of various degrees.

The boundary itself may provide the parameters for the Fourier transform as shown in Fig. 8.7. The parameterization of Fig. 8.7 gives the following series expansions:

$$\mathbf{x}(\overset{S}{p}) = \Sigma \mathbf{X}_k e^{jkw_0 \overset{S}{s}} \qquad w_o = 2\pi/P, \quad P = \text{perimeter} \tag{8.2}$$

where the discrete Fourier coefficients $\mathbf{X}_k$ are given by

$$\mathbf{X}_k = \frac{1}{P} \int_0^P \mathbf{x}(s) e^{-jkw_o s} ds \tag{8.3}$$

A common feature for the Fourier descriptors is that typically the general shape is given rather well by a few of the low-order terms in the expansion of the boundary curve. Properly parameterized, the coefficients are independent of size, translation, and rotation of the shape to be described. The descriptors do not lend themselves well to reconstruction of the boundary; for one thing, the resulting curve may not be closed if only a finite number of coefficients is used for the reconstruction.

The $\psi - s$ curve may be used as the basis for a Fourier transform shape description [Barrow and Popplestone 1971]. $\psi(s)$ is converted to $\phi(s)$: $\phi(s) = \psi(s) - 2\pi\ s/P$. This operation subtracts out the rising component. A number of shape-indicating numbers arise from taking the root-mean-square amplitudes of the Fourier components of $\phi(s)$, discarding phase information. The shape descriptors are again indicative of the "lobedness" of the shape.

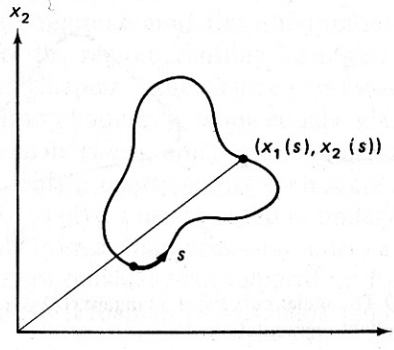


Fig. 8.7 Parameterization for Fourier Series Expansion.

8.2.5 Conic Sections

Polynomials are a natural choice for curve representation, and certain polynomials of degree 2 (namely, circles and ellipses) are closed curves and hence define regions. Circles may be represented with three parameters, ellipses by five, and general conics by six. Thus the coefficients or parameters of conic sections are terse representations. Conics are often good models for physical curves such as the edges of manufactured objects.

Conics are commonly used to represent general curves approximately [Paton 1970]. Conics have some annoying properties, however; an important one is the difficulty of producing a well-behaved conic from noisy data to be fitted. Unless one is careful in defining the error measure [Turner 1974], a "least-squared error" fit of a conic to data points yields a conic which is a nonintuitive shape or even of a surprising type (such as a hyperbola when an ellipse was expected). Conic representations and algorithms are explored in Appendix 1.

8.2.6 B-Splines

Interpolative techniques may be used to yield approximate representations. B-splines are a popular choice of piecewise polynomial interpolant. Introduced in computer aided design and computer graphics, these classes of curves provide adequate aesthetic content for much design and also have many useful analytic properties. Usually, the fact that the curves are "interpolating" is not very relevant. What is relevant is that they have predictable properties which make them easy to manipulate in image processing, that they "look good" to human beings, that they closely approximate curves of interest in nature, and so forth. Several schemes exist for constructing complex curves that are useful in geometric modeling, and detailed expositions are to be found in [deBoor 1978; Barnhill and Riesenfeld 1974]. The B-spline formulation is one of the simplest that still has properties useful for interactive modeling and the extraction from raw data.

B-splines are piecewise polynomial curves which are related to a *guiding polygon.* Cubic polynomials are the most frequently used for splines since they are the lowest order in which the curvature can change sign. An example of the relationship between the guiding polygon and its spline curve is shown in Fig. 8.8. Splines are useful in computer vision because they allow accurate, manipulable internal models of complex shapes. The models may be used to guide and monitor segmentation and recognition tasks. Interactive generation of complex shape models is possible with B-splines, and the fact that the complex spline curves have terse representations (as their guiding polygons) allows programs to manipulate them easily.

Spline approximations have good computational properties as well as good representational ones. First, they are *variation diminishing.* This means that the curve is guaranteed to "vary less" than its guiding polygon (many interpolation schemes have a tendency to oscillate between sample points). In fact, the curve is guaranteed to lie between the convex hull of groups of $n + 1$ consecutive points where n is the degree of the interpolating polynomial (Fig. 8.9.) The second advan-

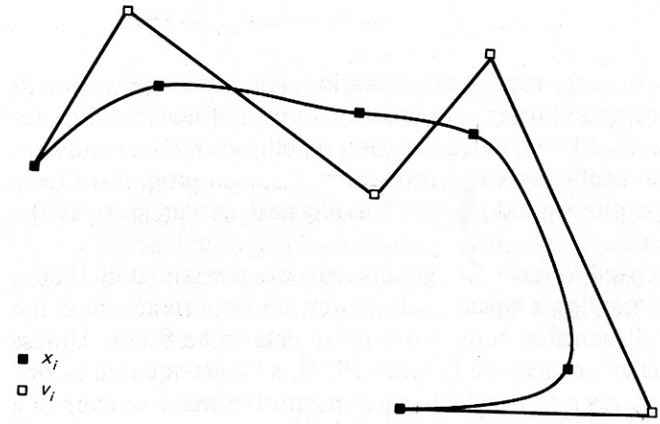


Fig. 8.8 A spline curve and its guiding polygon.

tage is that the interpolation is local; if a point on the guiding polygon is moved, the effects are intuitive and limited to nearby points on the spline. A third advantage is directly related to its use in vision; a technique for matching a spline-represented boundary curve against raw data is to search perpendicular to the spline for edges whose direction is parallel to the spline curve and location perpendicular to the spline curve. Perpendicular and parallel directions are computable directly from the parameters representing the spline.

B-Spline Mathematics

The interpolant through a given set of points $\mathbf{x}_i$, $i = 1, ..., n$ is $\mathbf{x}(s)$, a vector valued piecewise polynomial function of the parameter s; s changes uniformly between data points. For convenience, assume that $\mathbf{x}(i) = \mathbf{x}_i$, that is: s assumes integer values at data points, and $s = 1, ..., n$. Each *piece* of $\mathbf{x}(s)$ is a cubic polyno-

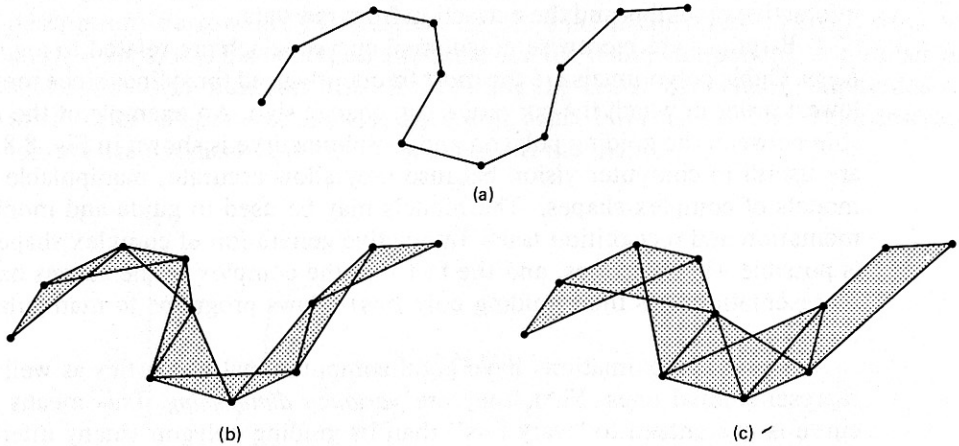


Fig. 8.9 The spline of degree n must lie in the convex hull formed by consecutive groups of $n + 1$ points. (a) $n = 1$ (linear). (b) $n = 2$ (quadratic). (c) $n = 3$ (cubic).

mial. Globally, $\mathbf{x}(s)$ has three orders of continuity across data points (i.e., up to continuity of second derivative: curvature). Formally, $\mathbf{x}(s)$ is defined as

$$\mathbf{x}(s) = \sum_{i=0}^{n+1} \mathbf{v}_i B_i(s) \tag{8.4}$$

The $\mathbf{v}_i$ are *coefficients* representing the curve $\mathbf{x}(s)$. They also turn out to be the vertices of the guiding polygon. They are a dual to the set of points $\mathbf{x}_i$; each can be derived from the other. The n data points $\mathbf{x}$ determine n $\mathbf{v}$'s. There are actually $n+2$ $\mathbf{v}$'s; the additional two coefficients are determined from *boundary conditions.* For example, if the curvature at the end points is to be 0,

$$\begin{aligned} \mathbf{v}_1 &= \frac{(\mathbf{v}_0 + \mathbf{v}_2)}{2} \\ \mathbf{v}_n &= \frac{(\mathbf{v}_{n-1} + \mathbf{v}_{n+1})}{2} \end{aligned} \tag{8.5}$$

Thus only n of the $n + 2$ coefficients are selectable.

The basis functions $B_i(s)$ are nonnegative and have a *limited support*, that is, each B_i is non-zero only for s between $i-2$ and $i+2$, as shown in Fig. 8.10. The limited support means that on a given span $(i, i+1)$ there are only four basis functions that are nonzero, namely: $B_{i-1}(s)$, $B_i(s)$, $B_{i+1}(s)$, and $B_{i+2}(s)$. Figure 8.11 shows this configuration. Thus, to calculate $\mathbf{x}(s_0)$ for some s_0, simply find in which span it resides, and then use only four terms in the summation (8.4), since there are only four basis functions which are non-zero there.

The basis functions $B_i(s)$ are, themselves, piecewise cubic polynomials and their definition depends on the relative size (in parameter space) of the spans under their support. If the spans are of uniform size (e.g., unity), then all the basis functions have the same *form* and are merely translates of each other. Moreover, each of the basis functions, on its nonzero support, is made of four pieces. So, in Fig. 8.11 in the span $(i, i+1)$ appear: the fourth piece of $B_{i-1}(s)$, the third piece of $B_i(s)$, the second piece of $B_{i+1}(s)$, and the first piece of $B_{i+2}(s)$. Call these pieces $C_{i,0}(s)$, ..., $C_{i,3}(s)$ respectively; then $x(s)$ on the interval $(i, i+1)$ is given by:

$$\begin{aligned} \mathbf{x}(s) = {} & C_{i-1,3}(s)\mathbf{v}_{i-1} + C_{i,2}(s)\mathbf{v}_i \\ & + C_{i+1,1}(s)\mathbf{v}_{i+1} + C_{i+2,0}(s)\mathbf{v}_{i+2} \end{aligned}$$

No matter what i is, $C_{i,j}$ will have the same shape; this property allows a simplification in calculations. Define four *primitive basis functions*, and interpolate along the curve by parameter shifting:

$$C_{i,j}(s) = C_j(s - i) \quad i = 0, \ldots, n+1; \qquad j = 0, 1, 2, 3 \tag{8.6}$$

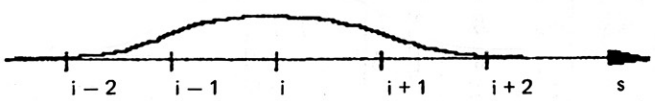


Fig. 8.10 Uniform B-spline: $B_i(s)$. Its support is non-zero only for s between $i-2$ and $i+2$.

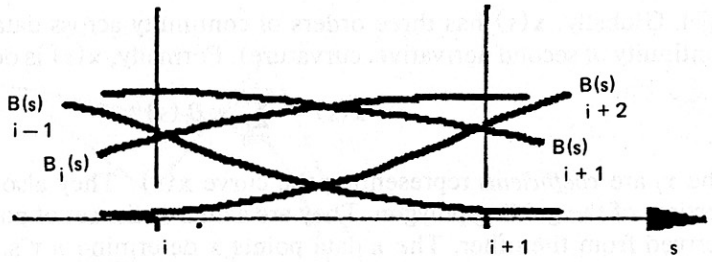


Fig. 8.11 The only four basis functions that are non-zero over the span $(i, i+1)$. Only the overlapping parts on this span are shown.

To find $\mathbf{x}(s_0)$, if s_0 is in the span $(i,\ i+1)$, use the formula:

$$\mathbf{x}(s) = \mathbf{v}_{i-1}C_3(s-i) + \mathbf{v}_i C_2(s-i) + \mathbf{v}_{i+1}C_1(s-i) + \mathbf{v}_{i+2}C_0(s-i) \tag{8.7}$$

where the $C_i(t)$ are given by:

$$C_0(t) = \frac{t^3}{6}$$

$$C_1(t) = \frac{-3t^3 + 3t^2 + 3t + 1}{6}$$

$$C_2(t) = \frac{3t^3 - 6t^2 + 4}{6}$$

$$C_3(t) = \frac{-t^3 + 3t^2 - 3t + 1}{6}$$

Formal derivations may be found in [Barnhill and Riesenfeld 1974; deBoor 1978].

Useful Formulae

The formulae may be simplified still further. $\mathbf{x}(s)$ is calculated in pieces (segments); define the segments $\mathbf{x}_i(t)$ where t ranges from 0 to 1. Then

$$\mathbf{x}_i(0) = \mathbf{x}_i \qquad \text{for } i = 1, \ldots, n-1$$

and

$$\mathbf{x}_{n-1}(1) = \mathbf{x}_n \tag{8.8}$$

In matrix notation, and explicitly calculating the definition of the cubic polynomials $C_i(t)$,

$$\mathbf{x}_i(t) = [t^3,\ t^2,\ t, 1][C][\mathbf{v}_{i-1},\ \mathbf{v}_i,\ \mathbf{v}_{i+1},\ \mathbf{v}_{i+2}]^T \tag{8.9}$$

where $[C]$ is the matrix:

$$\frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

The ith column in the matrix $[C]$ in Eq. (8.9) above is the coefficients of the cubic polynomial $C_i(t)$ $(i = 0, 1, 2, 3)$.

There is a distinction between *open* and *closed* curves. For open curves the boundary conditions must be used to solve for the two additional coefficients, as above. For closed curves, simply

$$\mathbf{v}_0 = \mathbf{v}_n \qquad \text{and} \qquad \mathbf{v}_{n+1} = \mathbf{v}_1 \tag{8.10}$$

The relation between the different $\mathbf{v}_i$ and $\mathbf{x}_i$ is summarized as follows. For open curves with zero curvature at the endpoints:

$$\begin{bmatrix} 6 & 0 & & & & & 1 \\ 1 & 4 & 1 & & & & \\ & \cdot & & & & & \\ & & \cdot & & & & \\ & & & \cdot & & & \\ & & & 1 & 4 & 1 & \\ & & & & 0 & 6 & \end{bmatrix} \begin{bmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{v}_{n-1} \\ \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{x}_{n-1} \\ \mathbf{x}_n \end{bmatrix}$$

and for closed curves:

$$\begin{bmatrix} 4 & 1 & & & & & 1 \\ 1 & 4 & 1 & & & & \\ & \cdot & & & & & \\ & & \cdot & & & & \\ & & & \cdot & & & \\ & & & 1 & 4 & 1 & \\ 1 & & & & 1 & 4 & \end{bmatrix} \begin{bmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{v}_{n-1} \\ \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{x}_{n-1} \\ \mathbf{x}_n \end{bmatrix} \tag{8.11}$$

Equation (8.10) gives the relationship between the points on the guiding polygon and the points on the spline. It may be derived from Eq. (8.9) with $t = 0$ (see exercises). To interpolate between these points, use a value of t between the extremes of 0 and 1. Choosing $t = k\,dt$ for $k = 0, \ldots, n$ where $n\,dt = 1$ and substituting into Eq. (8.9) yields

$$\mathbf{x}_i(k\,dt) = [(k\,dt)^3 (k\,dt)^2 (k\,dt) 1]\,[C]\,[\mathbf{v}_{i-1}, \mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}]^T \tag{8.12}$$

This can be decomposed [Wu et al. 1977; Gordon 1969] into the following equation.

$$\mathbf{x}_i(k\,dt) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} 1 & & & \\ 1 & 1 & & \\ 1 & 1 & 1 & \\ 1 & 1 & 1 & 1 \end{bmatrix}^k \begin{bmatrix} 6 & & & \\ -6 & 2 & & \\ 1 & -1 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} dt^3 \\ dt^2 \\ dt \\ 1 \end{bmatrix} [C] \begin{bmatrix} \mathbf{v}_{i-1} \\ \mathbf{v}_i \\ \mathbf{v}_{i+1} \\ \mathbf{v}_{i+2} \end{bmatrix} \tag{8.13}$$

The tangent at a curve is obtained by differentiation:

$$\mathbf{x}'_i(k\,dt) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} 1 & & \\ 1 & 1 & \\ 1 & 1 & 1 \end{bmatrix}^k \begin{bmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3dt^2 \\ 2dt \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_{i-1} \\ \mathbf{v}_i \\ \mathbf{v}_{i+1} \\ \mathbf{v}_{i+2} \end{bmatrix} \tag{8.14}$$

8.2.7 Strip Trees

In many computational problems there are space-time trade-offs. A nonredundant explicit representation for a general discrete curve, such as a chain code, is terse but may be difficult to use for certain computations. On the other hand, a representation for curves may take up much space but allow operations on those curves be very efficient. A representation with the latter property is *strip trees* [Ballard 1981]. Strip trees are closed under intersection and union operations, and these operations may be efficiently implemented.

A strip tree is a binary tree. The datum at each node is a eight-tuple, of which six entries define a strip (rectangle) and two denote addresses of the sons (if any). Thus each strip is defined by a six-tuple $S(\mathbf{x}_b, \mathbf{x}_e, \mathbf{w})$ as shown in Fig. 8.12. (Only five parameters are necessary to define an arbitrary rectangle, but the redundant representation proves useful in union and intersection algorithms to follow.)

The tree can be created from any curve by the following recursive procedure, which is very similar to Algorithm 8.1.

Algorithm 8.3: Making a Strip Tree

Find the smallest rectangle with a side parallel to the line segment $[\mathbf{x}_0, \mathbf{x}_n)$ that just covers all the points. This rectangle is the datum for the root node of a tree. Pick a point $\mathbf{x}_k$ that touches one of the sides of the rectangle. Repeat the above process for the two sublists $[\mathbf{x}_0, \ldots, \mathbf{x}_k)$ and $[\mathbf{x}_k, \ldots, \mathbf{x}_n)$. These become sons of the root node. Repeat the process until the approximation is accurate enough.

The half-open interval facilitates the computations to follow. In the example above the point $\mathbf{x}_k$ explicitly appears in both subtrees but implementationally need not be part of the left one. Figure 8.13 shows the strip tree construction process.

Intersecting Two Curves via Strip Trees

Consider what happens when a strip from one tree intersects a strip from another, as shown in Fig. 8.14. If the strips do not intersect, the underlying curves

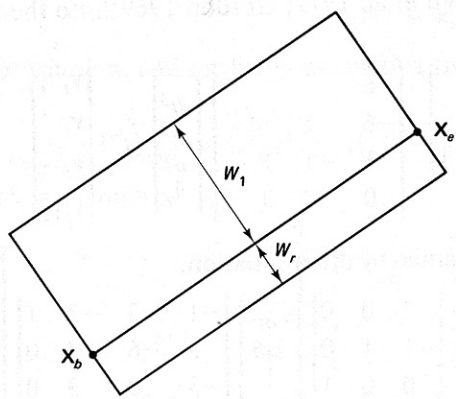


Fig. 8.12 Strip definition.

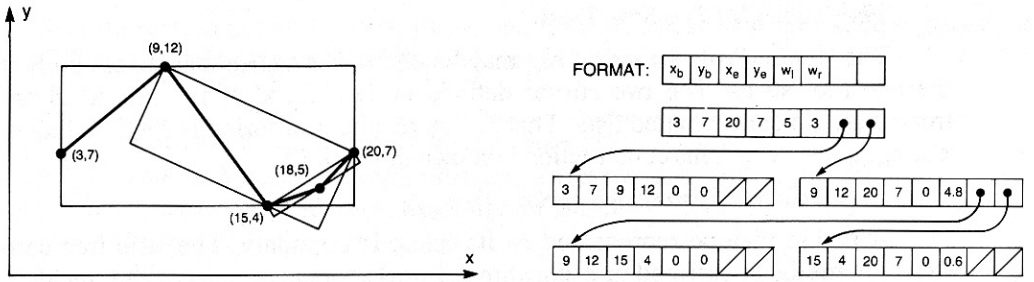


Fig. 8.13 Strip tree construction process.

do not intersect. If the strips do intersect, the underlying curves may or may not. To determine which, the computation may be applied recursively. At the leaf level of the tree defined as the *primitive* level, the problem can always be resolved.

Algorithm 8.4: Intersecting Two Strip Trees Representing Curves

Boolean Procedure TreeInt ($T1$, $T2$, L)
Begin
case intersection type of two strips $T1$ and $T2$ *of*
begin case
[primitive] *return* (true)
[null] *return* (false)
[possible] *If* $T2$ is the "fatter" strip
return (TreeInt($T1$,*L*Son($T2$) or TreeInt($T1$,*R*Son($T2$))
Else return (TreeInt(*L*Son($T1$),$T2$) or TreeInt(*R*Son($T1$),$T2$));
end case;
end;

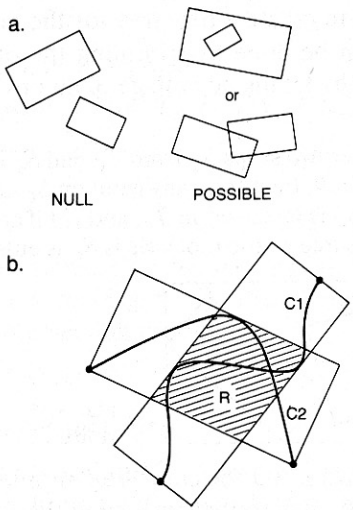


Fig. 8.14 Types of strip intersections. (a) Two kinds of intersections: NULL on the left; various POSSIBLE intersections on the right. (b) Under certain conditions the underlying curves must intersect.

The "Union" of Two Strip Trees

The "union" of two strip trees may be defined as a strip that covers both of the two root strips. The two curves defined by $[\mathbf{x}'_0, \ldots, \mathbf{x}'_n)$, $[\mathbf{x}''_0, \ldots, \mathbf{x}''_m)$ are treated as two concatenated lists. That is, the resultant ordering is such that $\mathbf{x}_0 = \mathbf{x}'_0$, $\mathbf{x}_{m+n+1} = \mathbf{x}''_m$. This construction is shown in Fig. 8.15.

Closed Curves Represented by Strip Trees

A region may be represented by its (closed) boundary. The strip-tree construction method described in Algorithm 8.3 works for closed curves and, incidentally, also for self-intersecting curves. Furthermore, if a region is not simply connected (has "holes") it can still be represented as a strip tree which at some level has connected primitives.

Many useful operations on regions can be carried out with strip trees. Examples are intersection between a curve and a region and intersecting two regions. Another example is the determination of whether a point is inside a region. Roughly, if any semi-infinite line terminating at the point intersects the boundary of the region an odd number of times, the point is inside. The implied algorithm is computationally simplified for strip trees in the following manner:

> *Point Membership Property.* To decide whether a point z is a member of a region represented by a strip tree, compute the number of nondegenerate intersections of the strip tree with any semi-infinite strip L which has $\|w\| = 0$ and emanates from z. If this number is odd, the point is inside the region.

This is because for clear intersections the underlying curves may intersect more than once but must intersect an odd number of times. A potential difficulty exists when the strip L is tangent to the curve. To overcome this difficulty in practice, a different L may be used.

Intersecting a Curve with a Region

The strategy behind intersecting a strip tree representing a curve with a strip tree representing a region is to create a new tree for the portion of the curve that overlaps the region. This can be done by trimming the original curve strip tree. Trimming is done efficiently by taking advantage of an obvious property of the intersection process:

> *Pruning Property:* Consider two strips S_C from T_C and S_a from T_a. If the intersection of S_C with T_a is null, then (a) if any point on S_C is inside T_a, the entire tree whose root strip is S_C is inside or on T_a, and (b) if any point on S_C is outside of T_a then the entire tree whose root strip is S_C is outside T_a.

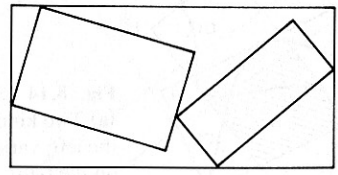


Fig. 8.15 Construction for "union" of strip trees representing two curves.

This leads to the Algorithm 8.5 for curve-region intersection using trees. If the curve strip is "fatter" (i.e., has more area), copy the node and resolve the in-

tersection at lower levels. In the converse case prune the tree sequentially by first intersecting the resultant pruned tree with the right region strip.

Algorithm 8.5: Curve–Region Intersection

```
comment A Reference Procedure returns a pointer;
reference procedure CurveRegionInt(T1,T2)
 begin
 A := T2;
 comment R is a global used by CRInt;
 return (CRInt(T1,T2));
 end;

reference procedure CRInt(T1,T2)
 begin
 begin Case StripInt(T1,T2) of
  [Null or Primitive]
   if intersection (T1,R,TRUE) = null then
    if Inside(T1,R) then return (T1)
    else return (null);
   else return (T1);
   [Possible] if T1 is "fatter" then
   begin
    NT := NewRecord;
    x_b(NT) := x_b (T);
    x_e(NT) := x_e (T);
    w_l(NT) := w_l(T);
    w_r(NT) := w_r(T);
    LSon(NT) := CRInt (LSon(T1),T2);
    RSon(NT) := CRInt (RSon(T1),T2);
    return(NT);
    end
   else comment T2 is "fatter"
    Return (CRInt(CRInt(T1,LSon(T2)),RSon(T2)));
   end;
end Case;
end;
```

The problem of intersecting two regions can be decomposed into two curve-region intersection problems (Fig. 8.16). Thus algorithm 8.5 can also be used to solve the region-region intersection problem.

8.3 REGION REPRESENTATIONS

8.3.1 Spatial Occupancy Array

The most obvious and quite a useful representation for a region on a raster is a membership predicate $p(x, y)$ which takes the value 1 when point (x, y) is in the